新人技術者のための

ロジカル・シンキング入門 第5回

リリースしたらテーブルが消えた！?

冴木 元



システムの記事　ビギナーズ

ここでは，コーディング規約について考える．コーディング規約（コーディング工程での管理方針）を定めておくことで，ソフトウェアの不ぐあいの発生をある程度抑えられる．ただし，関数や論理値の正負の定義，データ構造などをよく検討しないで規約を作ってしまうと，形骸化してしまう．単なる「おまじない」にならない規約の作りかた，そして規約の運用にあたっての考えかたを解説する．（編集部）

Dさんがリーダをつとめるプロジェクトは，1週間ほど前にリリースを終えたばかりです．ホッとしたのもつかの間，しごとを終えて帰宅しようとしたDさんのところに，リリース先のメーカに勤めているQさんから電話がかかってきました．リリースされたモジュールを組み込んだところ，正しい結果が得られず，ときどきシステムがハングアップするというのです．

まずDさんは，モジュールといっしょに提供したリリース・ノートを先方に確認してもらうことにしました．そして，わかりにくい点などを解説してみましたが，いずれも問題なく組み込んだはずだという答えが返ってきました．そこでDさんのプロジェクト・サーバに保管されているモジュールを動かしてみたのですが，先方が指摘するような現象は再現できません．

手詰まりになったDさんは，リリース先のQさんの会社におじゃましてモジュールの組み込み状況を見せてもらうことにしました．しかし，モジュールの引き数の渡しかたなどを見ても異常はみられません．「もしや，なんらかの処理系のバグかな」と思いかけたDさんの目に偶然飛びこんできたのは，ライブラリの呼び出し元である親ルーチンのソース・コードに書かれていた**図1**のような記述でした．

「あれ，おかしい．このテーブル[注1]は32ビット幅のはずだ．short型では正しく動かないんじゃないのか…．というか待てよ．そもそもこんな記述が呼び出し元にあるのに，なんでリンク・エラーが起きないんだ？」．

不安に思ったDさんは，プロジェクトのメンバに電話して，リリースしたソース・コードをもう一度確認してもらえないかと連絡してみることにしました．30分後に返ってきた答えは信じられないものでした．

「リリース前に使っていたテスト用関数のファイルにテーブルの実体をまちがえて書いてしまっていた．そのため，リリースしたライブラリにはこのテーブルの実体は含まれていない」．

なんと，ライブラリの中に含まれているべきテーブルの実体がリンクされていなかったというのです．しかも，たまたま呼び出し元で同名のテーブルがあったためにリン

```
static short DATA_LENGTH[40]=
{234, 123, ... ... .};
```

図1　処理系バグに思えたエラーの正体

リリース先で組み込み元のソース・コードを見ると，ライブラリ内部の記述と似たようなものが見つかった．こうしたものがあれば，リンク・エラーを起こすはず．起こさなかったということは…．テーブルだけ組み込まずにリリースしたというオチ．

注1：中身を書き換えず，プログラム起動時からの値が固定で用いられる配列のことをテーブルと呼ぶ．

KeyWord　テーブル，コード・レビュー，コーディング規約，ヘッダ・ファイル，define，プロトタイプ宣言，typedef，オブジェクト指向言語，外部変数，ウォータフォール・モデル

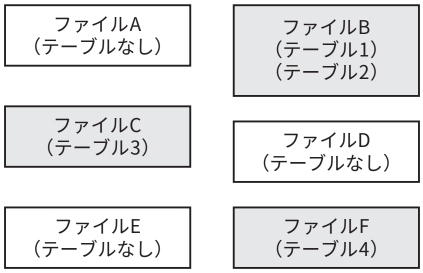


図2　バラバラで存在するテーブルは保守性が悪い

なぜなら，テーブルは外部参照していろいろなファイルで使うものであるため．どこになにがあるかわからないと，極めて保守性の悪いソース・コードになってしまう．

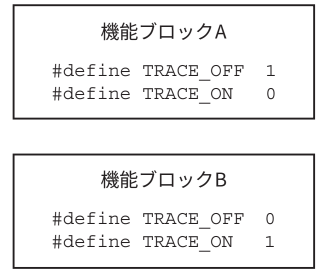


図3　論理値が機能ブロックによって逆という例

方針をまったく決めずに複数の人がコーディングしてしまうと，こういうことが起こりうる．防止するためには，ソース・コードの書きかたの規則に加え，必要な論理値は正負をあらかじめ決めておくというような対策が必要．

ク・エラーを起こさなかったので，リリース先でも気づかなかったということでした．

● テーブルの実体，論理値の統一に問題が

Dさんが平身低頭して直ちにリリースをやり直したのはいうまでもありません．

うなだれて帰ってきたDさんは翌日，プロジェクトのメンバを集めてソース・コード・レビューを行うことにしました．そこで判明したのは，リリースしたライブラリのソース・コードはテーブルの実体宣言が機能ブロックごとにいろいろなファイルにまたがっていて，あるテーブルの実体がどこにあるかはコードを書いた者にしかわからないという事実でした（**図2**）．さらに問題だったのは，散らばったテーブルのうち，いくつかは機能ブロック間で共通に使われているということでした．

ソース・コードをさらに細かく読み進めてみると，define文やプロトタイプ宣言のたぐいもあちこちにバラバラに存在していました．中にはプロトタイプ宣言なしで使われている関数もあるようでした．加えて問題だったのは，ある一つのdefine文は機能ブロックによって論理値が逆となっていたことです（**図3**）．

外部変数も実体があちこちにありました．外部変数を使用することは設計段階で必要と判断したものですが，その管理に問題があることは否めないようです．構造体や変数型のtypedefも散在していることがわかりました．

● 「おまじない」にならないためのコーディング規約作り

Dさんのような事態に陥った場合，読者のみなさんは，どういった解決策を考えるでしょうか．「そこまで自分はまぬけではない」と思う方もいるかもしれません．そういう方は，管理がボロボロのままライブラリを提供してしまう人たちを引き当ててしまった管理者のQさんになったつもりで考えてみてください．

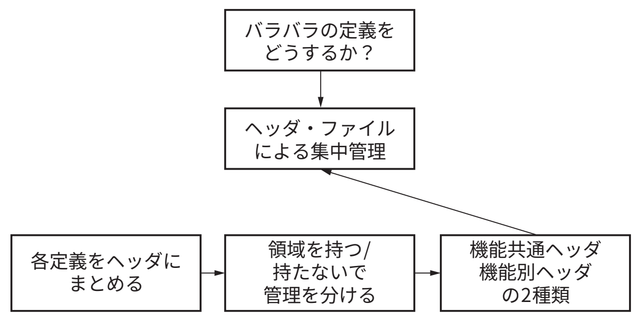


図4　コーディング規約の検討

バラバラな記述が混乱を生んでいる場合，それぞれをどのように整理すれば混乱がおさまるかをまず検討する．この過程をスキップしていきなり規約を作ると，形骸化してしまうことは避けられない．

まずだれでも考えるであろうことは，「再発防止のためには，ソース・コードになんらかの規約が必要なのではないか」ということでしょう．これについては，筆者も同じ意見です．しかし，コーディング規約のたぐいというものは，機械的に作って当てはめるだけですと次第に形骸化しかねません．そして最後には根拠の不明確な，単なる「おまじない」のようなものに陥る危険性があるのも事実です．たいせつなのは，ここで起こったような混乱を生じないためのしくみ作りとしてのコーディング規約作りだ，ということを念頭に置くことです（**図4**）．

そこでDさんといっしょに，上述の問題についてソース・コード規約を考えてみることにしましょう．手始めにDさんは，「定義箇所がバラバラ」なものを挙げてみました（**図5**）．

- define
- typedef（構造体や変数の型宣言）
- プロトタイプ宣言
- テーブル
- 外部変数

図5　バラバラに定義されると混乱のもとになるもの

例えば，テーブルがあちこちに存在してどこに何があるのかわからないような場合，冒頭の例で挙げたような定義ミスが起こりうる．

- define
- typedef（構造体や変数の型宣言）
- プロトタイプ宣言

} 領域を持たない

- テーブル
- 外部変数

} 領域を持つ

図6　「領域を持つ，持たない」で管理方針を分ける

ヘッダ・ファイルで定義を整理する場合，領域を持つものと持たないものは管理方法を分ける必要がある．なぜなら，前者はヘッダ・ファイルそのものに領域を確保してしまうと，ヘッダ・ファイルを複数のファイルでインクルードできなくなるから．

● ヘッダにまとめる管理方針

さて，これらを秩序立てて分類するためには，どのようにすればよいでしょうか．defineやtypedefなどは，要するに記述が秩序だてて行われていればよいのです．define用，typedef用については，以下のような方法が考えられます．

1）専用のヘッダ・ファイルにそれぞれ定義する

2）記述の順番を決めて，すべての定義を一つのヘッダにまとめる

1）はもっとも簡単な方法ですが，バラバラに記述されるのが混乱のもとなので，それを避けるためにdefineやtypedefを別のヘッダ・ファイルに分けない**2）**のような方法も採用できます．

1）または**2）**のいずれかを採用すれば，定義箇所がバラバラに散乱することは防げそうです．

● 領域を使うか否かで管理は異なる

「だが，ちょっと待てよ」と，Dさんはさらに考えます．「defineならともかく，テーブルなどはどうしよう？」．

なぜなら，テーブルは領域を持っているため，領域を持たない単なる型宣言とは分けて考えなければならないはずだからです．ヘッダ・ファイルを使ってテーブルの管理方針を考える場合，どのような方針が考えられるのでしょう．

「外部変数」や「テーブル」などは，変数であるため領域を持ちます（**図6**）．一方，defineなどはプリプロセッサが置き換えるだけのものであるため，それそのものが領域を持つわけではありません．

defineなどは領域を持たないので，ヘッダ・ファイルにおいても何の問題もありません．しかし，テーブルなどは領域を持つため，無秩序にヘッダに集めてしまうと多重定義が発生してリンク時にエラーを引き起こしてしまうことになります．

その一つの解決方法として，

- **外部変数やテーブルは，それらを定義するためのソース・ファイルを用意して，実体はそこに集めてしまう**

というものがあります．そして，

- **ヘッダ・ファイルにextern宣言を用意し，テーブルや外部変数が必要なファイルはそのヘッダ・ファイルをインクルードする**

ようにします．これらを方針として定めれば，テーブルや外部変数の参照が必要なファイルからのみアクセスできるようになります（**図7**）．

● 機能別ファイルと共通部品のくくり出し

商用開発されるようなソフトウェアは，かならず機能ブロックが分けられており，それぞれに担当が定められています．コーディング規約はここでどういう役割を果たせばよいのでしょうか．

Dさんは機能ブロックごとにヘッダ・ファイルを分ける

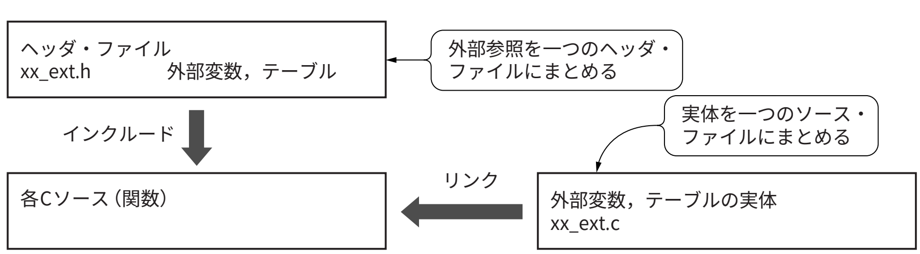


図7
ヘッダに領域を持たない解決策

一つの解決策として，領域を持つものの場合，領域の実体をCソース・ファイルにまとめて定義してしまう．そこへの参照宣言はヘッダ・ファイルにまとめる．

ことにしました．「この後，2次開発，3次開発を行う場合，機能ブロックごとにヘッダ・ファイルを更新してもらえば混乱しないはずだ」と考えたからです．

では，機能ごとに共通の部品はどうするのでしょう．機能ごとに開発を進めるにしても，共通部品というものはかならず出てきます．これらをじょうずにまとめていくのがうまい設計のやりかたなのですが，これをコーディング規約に反映させない手はなさそうです．

例えば，最初は機能ごとにルールに従ってソース・コードを作り込んでもらいます．そして，しかるべきタイミングで打ち合わせを行うなどして共通部品をくくり出し，それらを共通ヘッダに入れる，という方法で共通部分をくくりだすことができそうです．

つまり，ヘッダ・ファイルとして次の2種類を用意することになります．

- 各機能別のヘッダ・ファイル
- 各機能共通のヘッダ・ファイル

● 構成を決めたら順番も決める

次は`define`や`typedef`，プロトタイプ宣言などを実際にヘッダ・ファイルにまとめ直す段階です．

ヘッダ・ファイルの構成は前述したように，「記述の順番を決めて，すべての定義を一つのヘッダにまとめる」という方法が一つあります．ただし，この方法を用いると行数が多くなり過ぎたり，書き換えが頻繁に発生してしまいます．そこで，一つにまとめると混乱が生ずる場合は，`define`や`typedef`，プロトタイプ宣言ごとにヘッダ・ファイルを分けて，「専用のヘッダ・ファイルにそれぞれ定義する」という方法を採用します．

後者の場合，各ソース・ファイルにインクルードするヘッダ・ファイルの順番を決めておき，その後，「各ソース・ファイルは決められた順番でヘッダをインクルードする」ようにすると，混乱は生じなくなります．この構成ならあえて多重インクルードを行う必要はありません．つまり，コーディング規約として多重インクルードを禁じてしまうことができます．

上述のようなコーディング規約を用いていれば，Dさんのプロジェクトで起こったテーブルや`define`をめぐる大混乱は救えたことでしょう．

複数のソース・ファイルで利用されるようなさまざまな定義は，一定の規則に従ってヘッダ・ファイルを構成してそこで集中管理すれば，開発に秩序をもたらしうる性質のものです．以上をまとめると，**図8**，**図9**のようになります．

● 規約を考えるうえでのポイント

以上，コーディング規約の作成にあたっての具体例を挙げました．次に，一般的なコーディング規約を考えた場合に重要となる事柄について述べます．コーディング規約を考えるうえでは次の2点がポイントとなると考えます．

- 設計工程での検討もれは規約では救えない
- 規約は設計方針と調和させること

順に解説していきます．

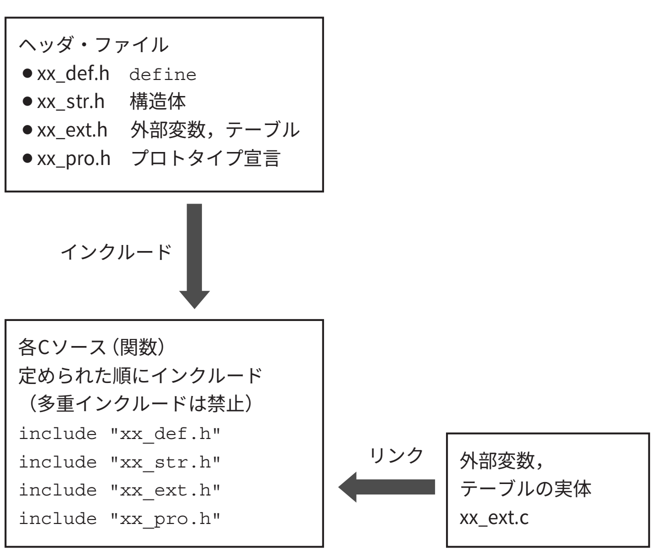


図8　ヘッダ・ファイルによる管理の全体像

種類ごとにヘッダ・ファイルを分けた例．規模の小さい場合，すべての定義を一つのヘッダ・ファイルにまとめたほうが見やすくなるかもしれない．

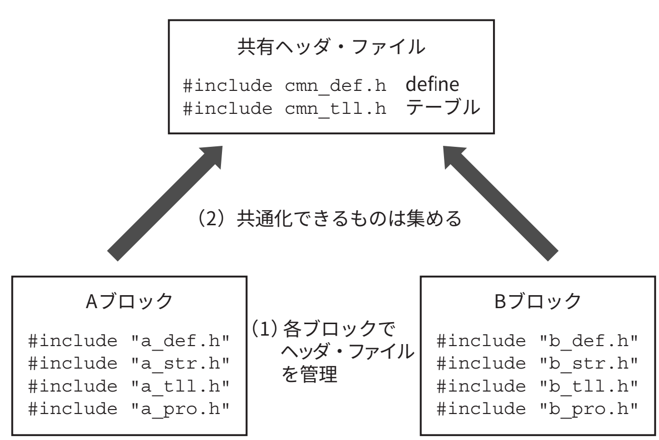


図9　共通部品化と機能分けのやりかた

図8のようなやりかたを各機能ブロックで行う場合，共通化したい部品がでることがある．それらは共通ヘッダにまとめて，各機能ブロックでインクルードできるようにする．

1）設計工程での検討もれは規約では救えない

最初に強調しておきたいのは，コーディング規約はあくまでコーディング工程の管理方針だということです（あまりにあたりまえのことのようだが…）．しかし言い換えれば，コーディングより上流工程における設計などに欠陥があって混乱が生じていた場合は，コーディング規約を定めたところで解決策にならない，ということです．

すなわち，設計工程での検討もれは，コーディング規約では救えないのです．救えないばかりか，そもそも守れない規則ができ上がってしまい，混乱は増すかもしれません．

そのような例としては，本連載の第2回（本誌2006年6月号，pp.60-66）で紹介したように外部変数が無秩序に使用されているプログラムが挙げられると思います．このようなプログラムにおいては，外部変数の管理方針を規約で定めたところでソース・コードの保守性はたいして向上しません．なぜなら，外部変数の無秩序な使用は，データ構造の再検討を抜きにしては解決のしようがないからです．

すなわち，システムにとって必要とされるデータ構造はどのようなものかを検討し，それを設計に反映させるようにしないと保守性は改善しないということです．

設計段階での検討がコーディング規約に先立って重要となる場合，**図9**で記述したような共通化と機能分けを例にとって考えてみることにします．

この共通化と機能分けという作業は，当然のことながら設計段階で検討すべき作業です．コーディング規約は，この設計段階での検討がスムーズに実装に移せるようにするくふうにほかなりません．

第3回（本誌2006年8月号，pp.102-108）で記述したように，システムの機能分けというものは設計の最初の段階で考えられる事がらです．そしてその機能分けに従って担当が割り振られ，工程表が組み立てられます．

そのような場合，もちろん各機能で用いられる共通モジュールなどが必要となってきます．ヘッダ・ファイルの共通化と機能分けの方針は，このような設計段階からの共通部品のくくり出しと機能分けを具体化するコーディング工程における作業なのです．

したがって，設計段階での部品共通化に問題がある場合は，後工程のコーディング規約でそれを救うことはできません．設計段階で検討すべき事がらをコーディング規約で救おうとしても救いきれないということです．

図3で，論理値が機能ブロックごとにバラバラな例を挙げました．各機能で共通に扱うような論理値は，ほんとうはコーディング前の設計の段階であらかじめ定めておいたほうがよいでしょう．

システムのデフォルト値はゼロが正しいか，非ゼロが正しいか，などということは絶対的な基準があるわけではありません．あるシステムではゼロをとればデフォルト値の意味になっていて，マイナス値はなんらかの異常を表すかもしれません．またあるシステムでは，‘1’がデフォルト値であって，‘2’，‘3’という値はなんらかの特別な場合を表すのかもしれません．

このようなことは，システム全体の要請を考慮しながら決定すべき事がらです．個人の好みはあっても，絶対的な正解というものはかならずしも存在しません．ただし，全体の方針で調和がとれていなかったら，確実に混乱のもとになります．**図9**では，共通ヘッダと機能別ヘッダを分ける方法の一例を紹介しています．このコーディング規約も，設計段階（コーディングの一歩手前の内部設計も含む）で検討した事がらを反映するようにしないと，プログラムの保守性は期待したほどは向上してくれません．

2）規約は設計方針と調和させること

コーディング規約は，設計という上流工程で定められた方針と調和したものでなければなりません（**図10**）．

例えば，Javaなどのオブジェクト指向言語でよく採用されるような，クラスごとにソース・ファイルを分けてソース・ファイルと同名のヘッダ・ファイルにクラスや構造体の定義を集めて構成していくという方針をとっている場合などは，本稿で述べたような管理方針をそのまま採用できません．そのような構成の場合，プロトタイプ宣言や構造体をクラスごとに別々のヘッダ・ファイルに入れないで共通化してしまうと，ソース・コードが書けないからです．オブジェクト指向を設計方針として採用し，そのためにオブジェクト指向言語を実装に用いると，結果としてコーデ

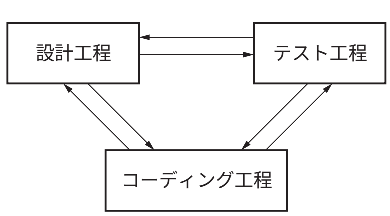


図10　各工程の方針は互いに調和させること

開発言語の設計思想が前工程の設計作業そのものに影響を与えるというのは，教科書的な知識からは本末転倒のように聞こえるが，実際の開発では見落とせないファクタとなっている．

ィング規約にも影響を及ぼすということです．

● 規則は趣旨を考えて運用する

なんらかの規則を定めた場合，それを適切に運用するには趣旨を考えることがとてもたいせつです．**図4**に示したようなソース・コードの管理方針の趣旨は，要するに「集中管理によって混乱を防ぐ」というものです．だとすれば，機能ごとにクラスを分けるコーディングを採用する場合も，同じ考えが応用できるはずです．すなわち，`define`やテーブルに関しては，集中管理する方針が採用できるということです．

そうではなく，「ヘッダ・ファイルは，構造体や`define`，プロトタイプ宣言，テーブルに分け，各関数共通で用いるものだ」と結論だけ覚えこんでいた人がいたとしたら，やっかいです．そのような人は，ここで解説したようなヘッダ・ファイルの管理方法を身につけたとしても，慣れないプログラム言語に初めて接した場合にはなかなか対応できないことになります．

逆に，設計とコーディング規約が合致する例を挙げてみましょう．データ構造は設計段階で検討すべき事がらですが，この方針が定まったうえで，それと調和のとれたコーディング規約を採用すると，コーディングの工程における実装がスムーズに進められます．

連載第2回では，データ構造を検討したうえで，スタティック・データ，ダイナミック・データ，テーブル，ローカル変数の4種類のデータがありうることを解説しました．これらを本稿で紹介したコーディング規約にあてはめればうまくいくはずです．

第2回では，スタティック・データとダイナミック・データについては，構造体にそれぞれまとめこんでしまう構成をとるようにしていました．つまり，上述のような規約を採用する場合は，構造体用ヘッダ・ファイルにまとめて定義すればよいことになります．

テーブルについても，今回のコーディング規約で管理方針が定められています．すなわち，Cソース・ファイルに実体を定義し，ヘッダ・ファイルに`extern`宣言をまとめればよいでしょう．

ローカル変数は領域の小さな変数に限定して関数ごとに定義されて用いられるもの，というのが第2回における方針でした．したがって，ヘッダ・ファイルで共通管理を行う必要はとくにありません．まとめると**表1**のようになります．

表1　データ構造とコーディング規約

設計工程で検討した事がらがスムーズに実装されるようにくふうするのがコーディング規約の趣旨．したがって，設計者が開発言語に不慣れであってはならない．

データ構造（第2回参照）	コーディング規約
スタティック・データ	構造体ヘッダに定義
ダイナミック・データ	
テーブル	テーブル用ヘッダに定義
ローカル変数	ヘッダでの分類はなし
—	`define`
	プロトタイプ宣言

● 「工程」というものの考えかた

以上，「設計工程での検討もれは規約では救えない」，「規約は設計方針と調和させること」という二つのポイントについて解説しました．

実のところ，これら二つは独立した方針というよりはむしろ互いに補い合うものです．なぜなら両者は，各工程の成果物をどのように定義し，期日を定めるかという，ソフトウェア開発管理においてもっともかなめとなる事がらに密接にかかわってくるからです．

工程ごとにその趣旨と成果物をよく検討することはたいせつです（**図11**）．このことは，いわゆる教科書的なウォータフォール・モデルを採用しないような開発であっても変わることはありません．

ただし，教科書的なウォータフォール・モデルにとらわれると見落としがちになるのが，「工程間の調和」という観点です．

教科書の中ならいざしらず，実際の組み込みシステムの開発の場面で「いかに実装するか」をまったく考えずに「何を作るか」を考えることはありません．実際にコーディングを始めないと問題点に気づかない，というのでは何のために設計工程があったのかわからなくなってしまいます．

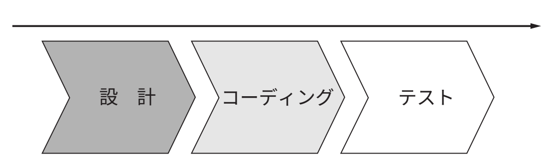


図11　工程には調和と独立性が求められる

教科書的なウォータフォール・モデルを採用しない場合であっても，開発チームを率いる人は，各工程の趣旨とその成果物を入念に考えておく必要がある．そのうえで各工程を調和させることもたいせつ．

組み込みシステムはハードウェア・リソースに強い制約があります．したがって，「いかに実装するか」をまったく考えずに「何を作るか」だけ考えてしまえば，実装不可能な仕様を定めてしまったことに後から気づくことにもなりかねません．仕様として定められた機能がいかに魅力的でも，現状のハードウェアで実装できないものであればアウトです．

したがって，ウォータフォール・モデルの基本的な考えかたとされる「whatを考えてからhowを考える」というのは，机上の空論であると少なくとも筆者は考えます．つねに後工程での作業も視野に入れながら，あらかじめ問題点を先取りし，一つ一つの工程における作業内容を確定できることが管理者に求められる資質といえるでしょう．

● 外部変数は最小限に絞り込んで

最後に，外部変数の使用について少し補足しておきます．

外部変数については，連載第2回でなぜそれが必要なのかを検討しました．また，データ構造の例としてはリードのみの外部変数であるテーブル以外を禁止してしまうやりかたを解説しました．

本稿では逆に，外部変数をとくに禁止しない手法を解説しています．しかしそうはいっても，外部変数がむやみやたらと使われるのはあまり望ましいことではありません．「必要性を吟味して，必要最小限のものに絞り込むことが望ましい」と考えられます．数が少なければ，頭に入れたうえでソース・コードをメンテナンスできるので，弊害が大きくなることはないと思われるからです．

外部変数の採用方針はコーディング規約の中で定める事がらです．しかし，外部変数を使うかどうか，また使うのであればどの程度なのかという問題は，実のところデータ構造をいかに定めるかという設計工程における検討と密接にかかわってきます．

繰り返しになりますが，外部変数については，設計工程とよく調和する形で管理方針を定め，必要最小限のものに絞り込むことが望ましいと思います．

さえき・はじめ

<筆者プロフィール>
冴木元．システム・エンジニア．刷り上った8月号の自分の記事を見てビックリ．なんとフローチャートがまちがっているではないか！ 筆者校正はやはりまじめにやるべきという戒めかもしれない．読者のみなさんすみません．